

L4 ARM-ISA

EECS 370 – Introduction to Computer Organization – Winter 2022

Outline

- ARM ISA Arithmetic and Logic
- ARM ISA memory instructions
- C -> Assembly



L4_1 ARM ISA: Arithmetic Logical Instructions

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

- Recognize the set of instructions for ARM Architecture (ISA) and be able to describe the operations and operands for instructions
 - LEGv8 subset
- Ability to create simple ARM assembly programs, e.g., using mathematical, logic, and memory operations

Resources

- Many resources on 370 website
 - <https://eeecs370.github.io/#resources>
 - ARMv8 references
- Discussion recordings
- Piazza
- Office hours

The screenshot shows the EECS 370 course website. On the left, there's a sidebar with links for Calendar, Lecture discussion, Assignments, Exams, Admin Requests, Schedule, Course Resources, Staff, Syllabus, Piazza, Files, Office Hours, Gradescope, and Canvas. The main content area has a header "EECS 370: Intro to Computer Organization" and "The University of Michigan, Winter 2022". Below this is a "Calendar" section with a weekly view from Jan 16 to 22, 2022. It lists events like "9am - 10:30am EECS 370 Section 001" and "10am - 11:30am EECS 370 Section 002". There's also a "register aliases" section for office hours.

EECS 370 - Introduction to Computer Organization

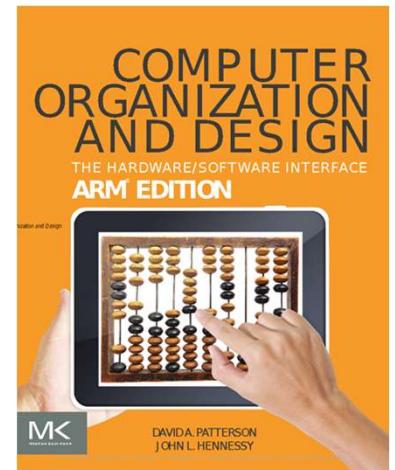
| EECS 370: GREEN CARD FOR LEGv8 | | | | | | |
|---------------------------------------|-------------------------|--|---------|-----------------------------|--|--|
| Arithmetic Operations | | Assembly code | | Semantics | | Comments |
| add | ADD Xd, Xn, Xm | Xs = Xd + Xm | | Xs = Xd + X7 | | |
| add & set flags | ADDS Xd, Xn, Xm | Xm | #immm12 | Xs = Xd + X7 | | |
| add signed word | ADDS Xd, Xn, Xm | #immm12 | | Xs = Xd + #19 | | 0 ≤ 12 bit unsigned ≤ 4095 |
| add immediate & set flags | ADDI9S Xd, Xn, #immm12 | | | Xs = Xd + #19 | | flags NZVC |
| subtract | SUB Xd, Xn, Xm | Xm | | Xs = Xd - Xm | | |
| subtract & set flags | SUBS Xd, Xn, Xm | Xm | #immm12 | Xs = Xd - X7 | | register-to-register |
| subtract immediate | SUBI9S Xd, Xn, #immm12 | | | Xs = Xd - #20 | | register-to-memory |
| subtract signed immediate & set flags | SUBIS Xd, Xn, #immm12 | Xm | | Xs = Xd - #20 | | 0 ≤ 12 bit unsigned ≤ 4095 |
| Data Transfer Operations | | Assembly code | | Semantics | | Comments |
| load register | LDUR Xd, [Xn, #immm9] | X2 = M[Xd, #181] | | X2 = M[Xn + #immm9] | | register-to-register |
| load signed word | LDURSW Xd, [Xn, #immm9] | X2 = M[Xd, #181] | | X2 = M[Xn + #immm9] | | double word load into Xd from Xn + #immm9, sign extend upper 32b |
| load byte | LDURB Xd, [Xn, #immm9] | X2 = M[Xd, #181] | | X2 = M[Xn + #immm9] | | word load to lower 32b of Xd from Xn + #immm9, zero extend upper 28b |
| store register | STUR Xd, [Xn, #immm9] | M[Xd] = X4 | | M[Xn + #immm9] = X4 | | double word store from Xd to Xn + #immm9 |
| store word | STURW Xd, [Xn, #immm9] | M[Xd] = X4 | | M[Xn + #immm9] = X4 | | word store from lower 32b of Xd to Xn + #immm9 |
| store half word | STURH Xd, [Xn, #immm9] | M[Xd] = X4 | | M[Xn + #immm9] = X4 | | by word, double word, or half word store from Xd to Xn + #immm9 |
| store byte | STURB Xd, [Xn, #immm9] | M[Xd] = X4 | | M[Xn + #immm9] = X4 | | byte load from least 8b of Xd to Xn + #immm9 |
| | | #immm9 = -259 to +258 | | | | -259 ≤ 9 bits signed immediate ≤ 4255 |
| move wide with zero | MOVZ Xd, #immm16, | LSL N | | X9 = 0, O, O, O, O | | |
| move wide with keep | MOVK Xd, #immm16, | LSL N | | X9 = x, x, x, x, x, x, x, x | | |
| | | X28 = SP; X29 = FP; X30 = LR; X31 = XR | | | | |
| register aliases | | | | | | Comments |
| | | | | | | zero extend then sign extend into the first (N = 0)/second (N = 16)/third (N = 32)/fourth (N = 48) 16b slot of Xd, without changing the other values (x's) |
| Logical Operations | | Assembly code | | Semantics | | Using C operations of & ^ < < > > |
| and | AND Xd, Xn | Xm | #immm12 | X3 = X2 & X7 | | bit-wise AND |
| and immediate | ANDI Xd, Xn, #immm12 | Xm | | X3 = X2 & #19 | | bit-wise AND with 0 ≤ 12 bit unsigned ≤ 4095 |
| inclusive or | ORR Xd, Xn | Xm | #immm12 | X3 = X2 X7 | | bit-wise OR |
| inclusive or immediate | ORRI Xd, Xn, #immm12 | Xm | | X3 = X2 #11 | | bit-wise OR with 0 ≤ 12 bit unsigned ≤ 4095 |
| exclusive or | EOR Xd, Xn | Xm | #immm12 | X3 = X2 ^ X7 | | bit-wise EOR |
| exclusive or immediate | EORI Xd, Xn, #immm12 | Xm | | X3 = X2 ^ #10 | | bit-wise EOR with 0 ≤ 12 bit unsigned ≤ 4095 |
| logical shift left | LSL Xd, Xn | #immm6 | | X1 = X2 << #10 | | shift left by a constant ≤ 63 |
| logical shift right | LSR Xd, Xn | #immm6 | | X1 = X2 >> #10 | | shift right by a constant ≤ 63 |
| move wide with keep | MOVK Xd, #immm16, | LSL N | | X9 = x, x, x, x, x, x, x, x | | place a 16b (#immm16) into the first (N = 0)/second (N = 16)/third (N = 32)/fourth (N = 48) 16b slot of Xd, without changing the other values (x's) |
| register aliases | | | | | | Using C operations of & ^ < < > > |
| | | | | | | third (N = 32)/fourth (N = 48) 16b slot of Xd, without changing the other values (x's) |
| Logical Operations | | Assembly code | | Semantics | | Using C operations of & ^ < < > > |
| and | AND Xd, Xn | Xm | #immm12 | X5 = X2 & X7 | | bit-wise AND |
| and immediate | ANDI Xd, Xn, #immm12 | Xm | | X5 = X2 & #19 | | bit-wise AND with 0 ≤ 12 bit unsigned ≤ 4095 |
| inclusive or | ORR Xd, Xn | Xm | #immm12 | X5 = X2 X7 | | bit-wise OR |
| inclusive or immediate | ORRI Xd, Xn, #immm12 | Xm | | X5 = X2 #11 | | bit-wise OR with 0 ≤ 12 bit unsigned ≤ 4095 |
| exclusive or | EOR Xd, Xn | Xm | #immm12 | X5 = X2 ^ X7 | | bit-wise EOR |
| exclusive or immediate | EORI Xd, Xn, #immm12 | Xm | | X5 = X2 ^ #10 | | bit-wise EOR with 0 ≤ 12 bit unsigned ≤ 4095 |
| logical shift left | LSL Xd, Xn | #immm6 | | X1 = X2 << #10 | | shift left by a constant ≤ 63 |
| logical shift right | LSR Xd, Xn | #immm6 | | X5 = X3 >> #20 | | shift right by a constant ≤ 63 |
| Unconditional branches | | Assembly code | | Semantics | | Using C operations of & ^ < < > > |
| branch | B #immm26 | | | goto PC + #1200 | | PC relative branch to PC + 26b offset; -2^25 ≤ #immm26 ≤ 2^25-1, 4b instruction |
| branch to register | BR Xt | | | target in Xt | | Xt contains a full 64b address |
| branch with link | BL #immm26 | | | X30 = PC + 4; PC + #11000 | | PC relative branch to PC + 26b offset; 16 million instructions; X30 = LR contains return from subroutine address |
| | | | | | | Also known as Jumps |
| | | | | | | PC relative branch to PC + 26b offset; -2^25 ≤ #immm26 ≤ 2^25-1, 4b instruction |

ARMs and LEGs

Subset of ARM.

ARM ISA

- ARMv8 is the 64-bit version—all registers are 64 bits wide
- Addresses are calculated to be 64 bits too
- **BUT: Instructions are 32 bits**
- We use a (small) subset of the v8 ISA used in P+H
- It is referred to as the LEGv8 in keeping with the body part theme!



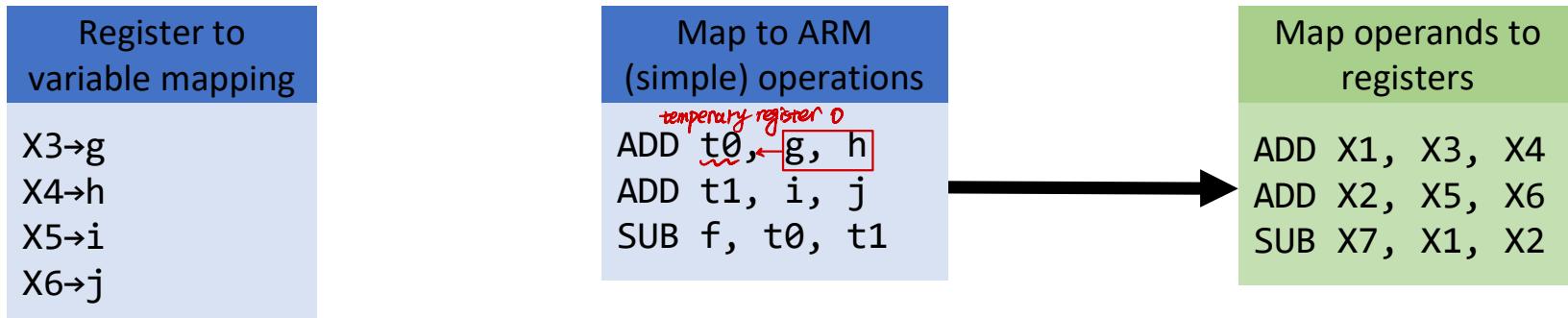
ARM Instruction Set—LEGv8 subset



- The main types of instructions fall into the familiar classes we saw with LC-2K:
 1. Arithmetic
 - Add, subtract, multiply (not in LEGv8)
 2. Data transfer
 - Loads a stores—LDUR (load unscaled register), STUR, etc.
 3. Logical
 - AND, ORR, EOR, etc.
 - Logical shifts, LSL, LSR
 4. Conditional branch
 - CBZ, CBNZ, B.cond
 5. Unconditional branch (jumps)
 - B, BR, BL

LEGv8 Arithmetic Instructions

- Format: three operand fields
 - Destination register usually the first one – *check instruction format*
 - $\text{ADD } X3, X4, X7$ – Think $\text{ADD } X3 = X4, X7$
 - LC-2K generally has the destination on the right!!!!
 - e.g., `add 1 2 3 // r3 = r1 + r2`
- C-code example: $f = (g + h) - (i + j)$



LEGv8 R-instruction Fields

| opcode | Rm | shamt | Rn | Rd |
|---------|--------|--------|--------|--------|
| 11 bits | 5 bits | 6 bits | 5 bits | 5 bits |

- Register-to-register operations
- Consider ADD X3, X4, X7
 - $R[Rd] = R[Rn] + R[Rm]$
 - Rd = X3, Rn = X4, Rm = X7
- Rm = second register operand
- shamt = shift amount
 - not used in LEG for ADD/SUB and set to 0
- Rn = first register operand
- Rd = destination register
- ADD opcode is 10001011000, what are the other fields?

LEGv8 R-instruction Fields

| opcode | Rm | shamt | Rn | Rd |
|---------|--------|--------|--------|--------|
| 11 bits | 5 bits | 6 bits | 5 bits | 5 bits |

- Register-to-register operations
- Consider ADD X3, X4, X7
 - $R[Rd] = R[Rn] + R[Rm]$
 - Rd = X3, Rn = X4, Rm = X7
- Rm = second register operand
- shamt = shift amount
 - not used in LEG for ADD/SUB and set to 0
- Rn = first register operand
- Rd = destination register
- ADD opcode is 10001011000, what are the other fields?

LEGv8 Arithmetic Operations

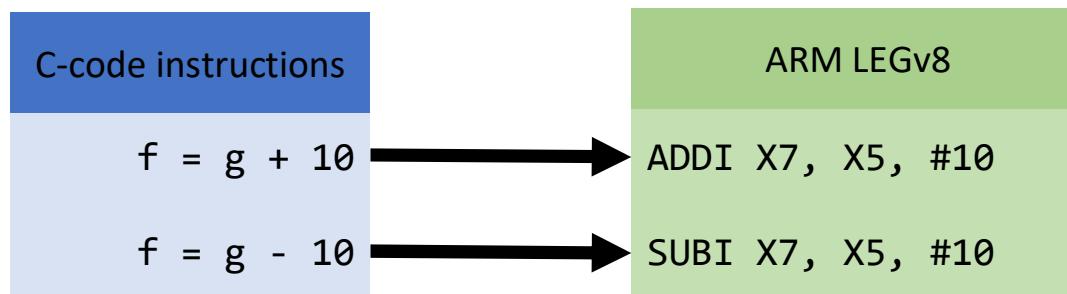
- Machine State—more on this concept as our understanding evolves
 1. Registers: 32 registers, 64-bit wide. X31 aliased to XZR which is always 0
- cannot use as a destination
 2. PC—Program counter
 3. FLAGS: NZVC – record the results of (arithmetic) operations
Negative, Zero, Overflow, Carry—*not present in LC2K*

| Category | Instruction | Example | Meaning | Comments |
|------------|----------------------------------|------------------|--------------|--|
| Arithmetic | add | ADD X1, X2, X3 | X1 = X2 + X3 | Three register operands |
| | subtract | SUB X1, X2, X3 | X1 = X2 - X3 | Three register operands |
| | add immediate | ADDI X1, X2, 20 | X1 = X2 + 20 | Used to add constants |
| | subtract immediate | SUBI X1, X2, 20 | X1 = X2 - 20 | Used to subtract constants |
| | add and set flags | ADDS X1, X2, X3 | X1 = X2 + X3 | Add, set condition codes |
| | subtract and set flags | SUBS X1, X2, X3 | X1 = X2 - X3 | Subtract, set condition codes |
| | add immediate and set flags | ADDIS X1, X2, 20 | X1 = X2 + 20 | Add constant, set condition codes |
| | subtract immediate and set flags | SUBIS X1, X2, 20 | X1 = X2 - 20 | Subtract constant, set condition codes |

I-instruction fields

| opcode | immediate | Rn | Rd |
|---------|--------------------------------|--------|--------|
| 10 bits | 12 bits <i>(unsigned)</i> | 5 bits | 5 bits |

- Format: second source operand can be a register or Immediate—a constant in the instruction itself
 - e.g., ADDI X3, X4, + #10
- Format: 12 bits for immediate constants 0-4095
- Do not need negative constants because we have SUBI



LEGv8 Logical Instructions

- Logical operations are *bit-wise*
- For example assume
 - $X9 = 11111111\ 11111111\ 11111111\ 00000000\ 00000000\ 00000000\ 00001101\ 11000000$
 - $X14 = 00000000\ 00000000\ 11011010\ 00000000\ 00000000\ 00000000\ 00111100\ 00000000$
 AND X2, X14, X9 would result in
 - $X2 = 00000000\ 00000000\ 11011010\ 00000000\ 00000000\ 00000000\ 00001100\ 00000000$
- AND and OR correspond to C operators `&` and `|`
- For immediate fields the 12-bit constant is padded with zeros to the left—zero extended

| Category | Instruction | Example | Meaning | Comments |
|----------|------------------------|---------|--------------|--|
| Logical | and | AND | $X1, X2, X3$ | $X1 = X2 \& X3$ Three reg. operands; bit-by-bit AND |
| | inclusive or | ORR | $X1, X2, X3$ | $X1 = X2 X3$ Three reg. operands; bit-by-bit OR |
| | exclusive or | EOR | $X1, X2, X3$ | $X1 = X2 ^ X3$ Three reg. operands; bit-by-bit XOR |
| | and immediate | ANDI | $X1, X2, 20$ | $X1 = X2 \& 20$ Bit-by-bit AND reg. with constant |
| | inclusive or immediate | ORRI | $X1, X2, 20$ | $X1 = X2 20$ Bit-by-bit OR reg. with constant |
| | exclusive or immediate | EORI | $X1, X2, 20$ | $X1 = X2 ^ 20$ Bit-by-bit XOR reg. with constant |
| | logical shift left | LSL | $X1, X2, 10$ | $X1 = X2 \ll 10$ Shift left by constant |
| | logical shift right | LSR | $X1, X2, 10$ | $X1 = X2 \gg 10$ Shift right by constant |

LEGv8 Shift Logical Instructions

| opcode | Rm | shamt | Rn | Rd |
|---------|--------|--------|--------|--------|
| 11 bits | 5 bits | 6 bits | 5 bits | 5 bits |

Last letter tells you the direction

- LSR X6, X23, #2
 $\text{Logical shift Right}$
 $X23 = 11111111 \ 11111111 \ 11111111 \ 00000000 \ 00000000 \ 00000000 \ 11011010 \ 000000\textcolor{red}{10}$
 $X6 = \textcolor{blue}{00}111111 \ \textcolor{blue}{11}111111 \ \textcolor{blue}{11}111111 \ \textcolor{blue}{11}000000 \ \textcolor{blue}{00}000000 \ \textcolor{blue}{00}000000 \ \textcolor{blue}{00}110110 \ \textcolor{blue}{10}000000$
- C-code equivalent : $X6 = X23 \gg 2;$
- Question: LSL X6, X23, #2 ?
 - What register gets modified?
 - What does it contain after executing the LSL instruction?
- In shift operations Rm is always 0—shamt is 6-bit unsigned

Shifting right by one bit -> divide by 2
 Shifting left by one bit -> multiple by 2

LEGv8 Shift Logical Instructions

| opcode | Rm | shamt | Rn | Rd |
|---------|--------|--------|--------|--------|
| 11 bits | 5 bits | 6 bits | 5 bits | 5 bits |

- LSR X6, X23, #2
 $X23 = 11111111 \ 11111111 \ 11111111 \ 00000000 \ 00000000 \ 00000000 \ 11011010 \ 00000010$
 $X6 = 00111111 \ 11111111 \ 11111111 \ 11000000 \ 00000000 \ 00000000 \ 00110110 \ 10000000$
- C-code equivalent : $X6 = X23 \gg 2;$
- Question: LSL X6, X23, #2 ?
 - What register gets modified?
 - What does it contain after executing the LSL instruction?

LSL X6, X23, #2
 $X23 = 11111111 \ 11111111 \ 11111111 \ 00000000 \ 00000000 \ 00000000 \ 11011010 \ 00000010$
 $X6 = 11111111 \ 11111111 \ 11111100 \ 00000000 \ 00000000 \ 00000011 \ 01101000 \ 00001000$

- In shift operations Rm is always 0—shamt is 6-bit unsigned

Shifting right by one bit -> divide by 2
 Shifting left by one bit -> multiple by 2

Pseudo Instructions

ARM ISA

- Instructions that use a shorthand “mnemonic” that expands to an assembly instruction
 - Exception to the "1-1 correspondence between assembly and MC" rule
- Example:
 - `MOV X12, X2` // the contents of X2 copied to X12 – X2 unchanged
- This gets expanded to:
 - `ORR X12, XZR, X2`
- What alternatives could we use instead of ORR?

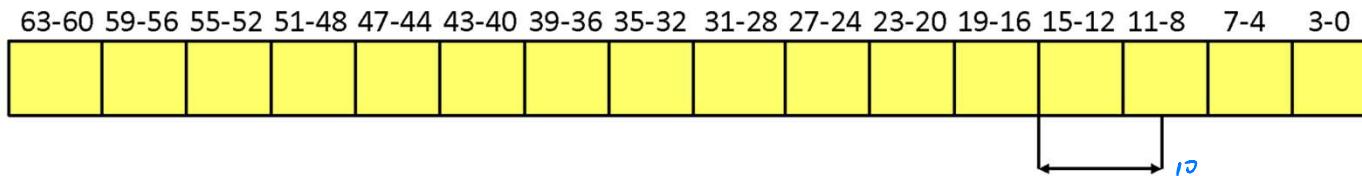
Pseudo Instructions



- Instructions that use a shorthand “mnemonic” that expands to an assembly instruction
 - Exception to the "1-1 correspondence between assembly and MC" rule
- Example:
 - `MOV X12, X2` // the contents of X2 copied to X12 – X2 unchanged
- This gets expanded to:
 - `ORR X12, XZR, X2`
- What alternatives could we use instead of ORR? – `ADD X12, XZR, X2`

LEGv8 Assembly Example #1

- Show the C and LEGv8 assembly for extracting the value in bits 15:10 from a 64-bit integer variable

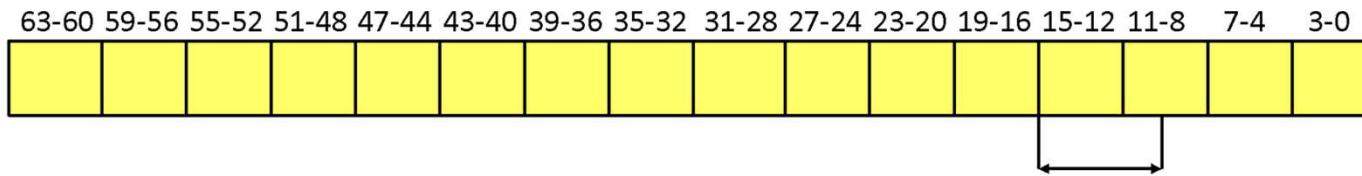


Assume the variable is in register X1

Want these bits

LEGv8 Assembly Example #1

- Show the C and LEGv8 assembly for extracting the value in bits 15:10 from a 64-bit integer variable

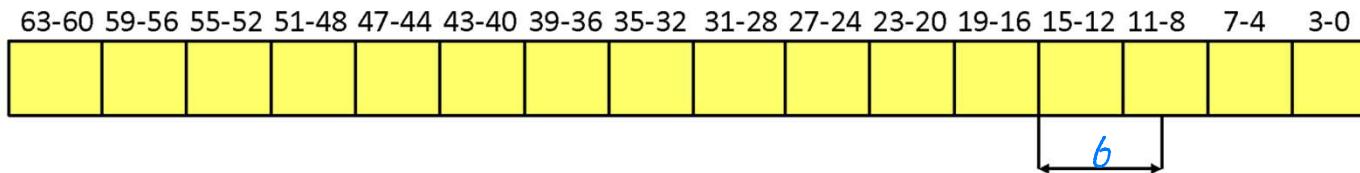


Assume the variable is in register X1

Want these bits

LEGv8 Assembly Example #1

- Show the C and LEGv8 assembly for extracting the value in bits 15:10 from a 64-bit integer variable



Assume the variable is in register X1

Want these bits

| C-code instructions | ARM LEGv8 |
|---------------------------------|---------------------------------|
| <code>x = x >> 10;</code> | <code>LSR X1, X1, #10</code> |
| <code>x = x & 0x3F</code> | <code>ANDI X1, X1, #0x3F</code> |

mask operation

*we need 11111
3 F*

L4 :

30:04

L4_2 ARM ISA: Memory Instructions

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

- Recognize the set of instructions for ARM Architecture (ISA) and be able to describe the operations and operands for instructions
- Ability to create simple ARM assembly programs, e.g., using mathematical, logic, and memory operations

Word vs Byte Addressing

- A **word** is a collection of bytes
 - Exact size depends on architecture
 - in LC-2K and ARM, 4 bytes
 - **Double word** is 8 bytes
- LC-2K is **word addressable**
 - Each address refers to a particular **word** in memory
 - Want to move forward one int? Increment address by **one**
 - Want move forward one char? Uhhh... no
- ARM (and most modern ISAs) is **byte addressable**
 - Each address refers to a particular **byte** in memory
 - Want to move forward one int? Increment address by **four**
 - Want to move forward one char? Increment address by **one**

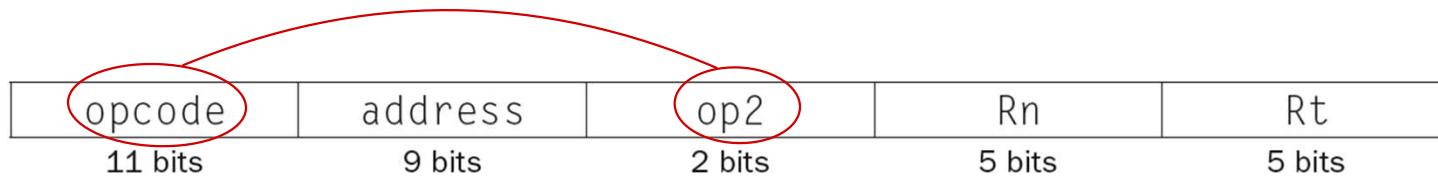
LEGv8 Memory Instructions

- Employs base + displacement addressing mode
 - Base is a register
 - Displacement is 9-bit signed immediate
 - Load signed word will sign extend to 64 bits
 - Load half and load byte will zero extend

? {

| Category | Instruction Example | Meaning | Comments |
|----------|---------------------|--|---|
| ? | load register | LDUR X1, [X2,40] X1 = Memory[X2 + 40] | Doubleword from memory to register |
| | store register | STUR X1, [X2,40] Memory[X2 + 40] = X1 | Doubleword from register to memory |
| | load signed word | LDURSW X1,[X2,40] X1 = Memory[X2 + 40] | Word from memory to register <i>take 32-bit value from memory, extends it to 64 bits and load it to register</i> |
| | store word | STURW X1, [X2,40] Memory[X2 + 40] = X1 | Word from register to memory |
| | load half | LDURH X1, [X2,40] X1 = Memory[X2 + 40] | Halfword memory to register |
| | store half | STURH X1, [X2,40] Memory[X2 + 40] = X1 | Halfword register to memory |
| | load byte | LDURB X1, [X2,40] X1 = Memory[X2 + 40] | Byte from memory to register |
| | store byte | STURB X1, [X2,40] Memory[X2 + 40] = X1 | Byte from register to memory |
| | move wide with zero | MOVZ X1,20, LSL 0 X1 = 20 or 20 * 2 ¹⁶ or 20 * 2 ³² or 20 * 2 ⁴⁸ | Loads 16-bit constant, rest zeros |
| | move wide with keep | MOVK X1,20, LSL 0 X1 = 20 or 20 * 2 ¹⁶ or 20 * 2 ³² or 20 * 2 ⁴⁸ | Loads 16-bit constant, rest unchanged |

D-Instruction fields



- Data transfer
- opcode and op2 define data transfer operation
 - address is the 9-bit signed immediate displacement
- Rn is the base register
- Rt is the destination (loads) or source (stores)
- More complicated modes are available in full ARMv8

LEGv8 Memory Instructions

- Registers are 64 bits wide
- But sometimes we want to deal with non-64-bit entities
 - e.g. ints (32 bits), chars (8 bits)
- When we load smaller elements from memory, what do we set the other bits to?
 - Option A: set to zero – LEGv8 instructions LDURH, LDURB



- Option B: sign extend – LEGv8 instruction LDURSW



Load Instruction Sizes

How much data is retrieved from memory at the given address?

- LDUR X3, [X4, #100]
 - Load (unscaled) to register—retrieve a double word (64 bits) from address (X4+100)
- LDURH X3, [X4, #100]
 - Load halfword (16 bits) from address (X4+100) to the low 16 bits of X3—top 48 bits of X3 are set zero
- LDURB X3, [X4, #100]
 - Load byte (8 bits) from address (X4+100) and put in the low 8 bits of X3—zero extend the destination register X3 (top 56 bits)
- What about loading words?
- LDURSW X3, [X4, #100]
 - retrieve a word (32 bits) from address (X4+100) and put in lower half of X3—top 32 bits of X3 are sign extended

LEGv8 Data Transfer Instructions--Stores

- Store instructions are simpler—there is no sign/zero extension to consider
- **STUR X3, [X4, #100]**
 - Store (unscaled) register—write the **double word** (64 bits) in register X3 to the 8 bytes at address (X4+100)
- **STURW X3, [X4, #100]**
 - Store word—write the **word** (32 bits) in the low 4 bytes of register X3 to the 4 bytes at address (X4+100)
- **STURH X3, [X4, #100]**
 - Store **half word**—write the half word (16 bits) in the low 2 bytes of register X3 to the 2 bytes at address (X4+100)
- **STURB X3, [X4, #100]**
 - Store **byte**—write the least significant byte (8 bits) in register X3 to the byte at address (X4+100)

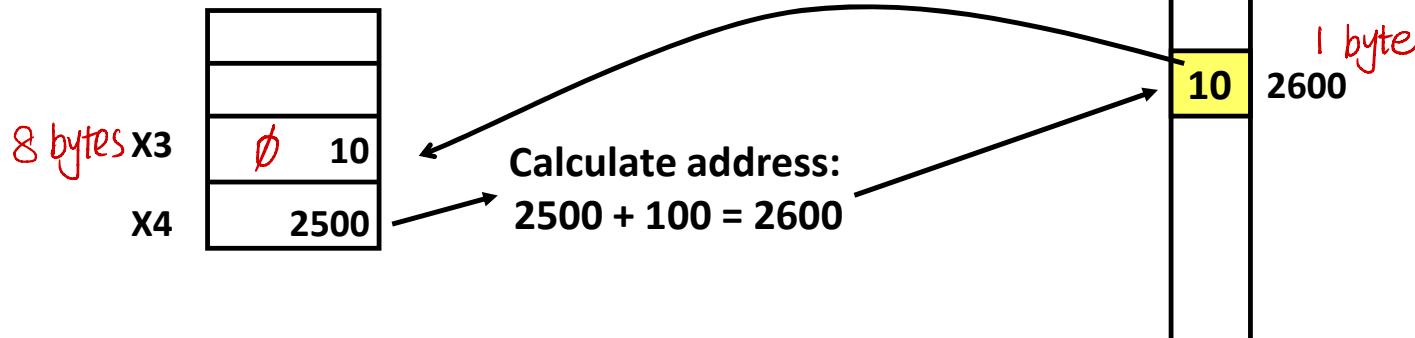
Load Instruction in Action

- LDURB X3, [X4, #100] // load byte

Retrieves 8-bit value from memory location (X4+100) and puts the result into X3. The other 56 most significant bits are 0—zero extended

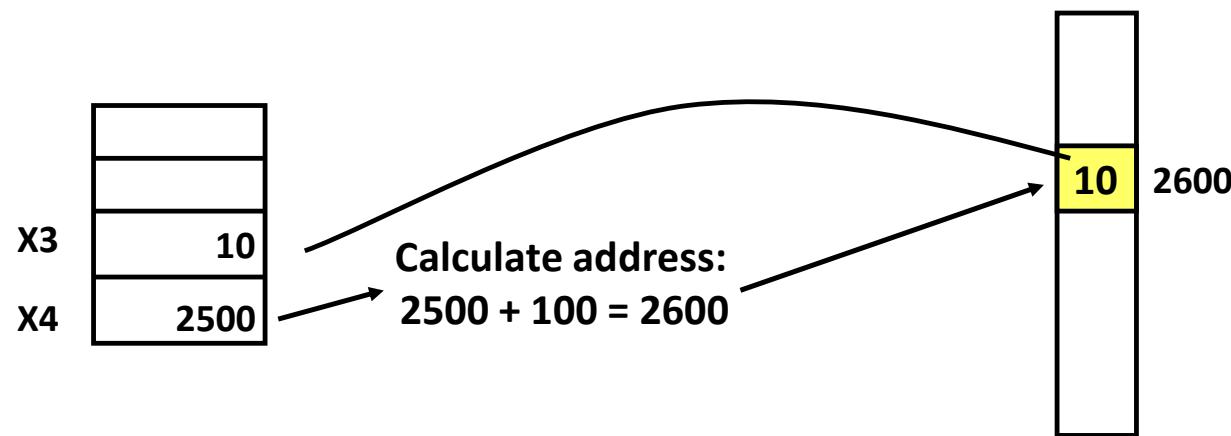
since unsigned

$$64 - 8 = 56$$



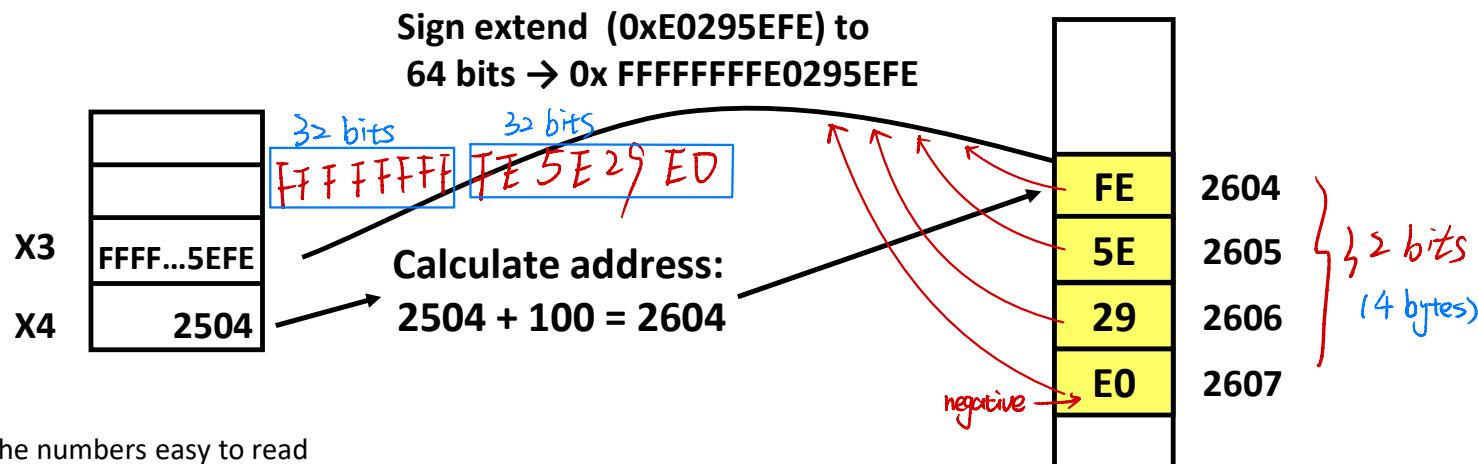
Load Instruction in Action

- LDURB X3, [X4, #100] // load byte
Retrieves 8-bit value from memory location ($X4+100$) and puts the result into X3. The other 56 most significant bits are 0—zero extended



Load Instruction in Action – Example #2

- LDURSW X3, [X4, #100] // load signed word
 Retrieves 4-byte value from memory location (X4+100) and
 puts the result into X3 (sign extended)

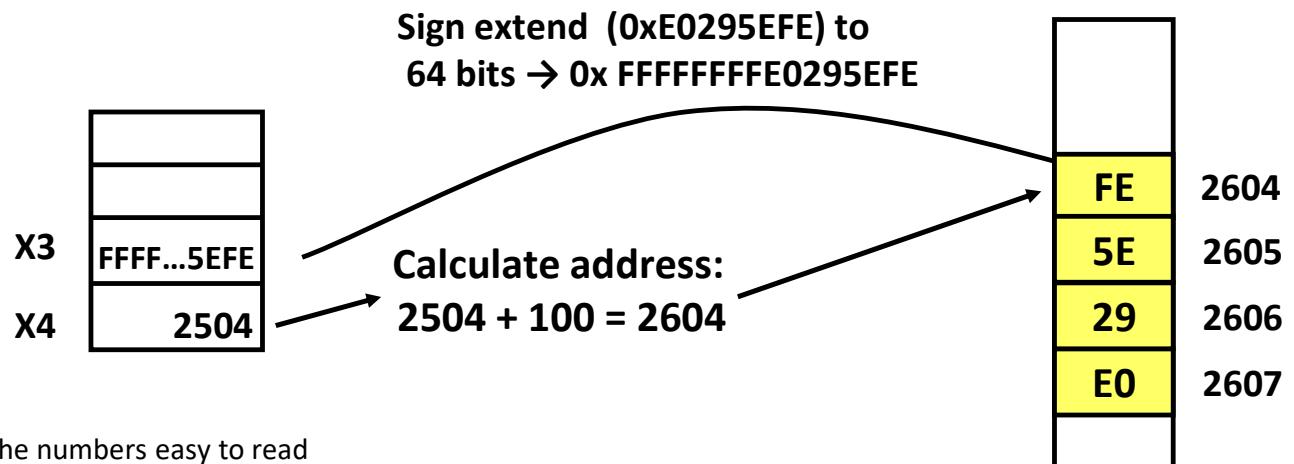


Example mixes decimal and hex to keep the numbers easy to read

Recall that E = 1110 and thus is treated as a negative 2's complement number

Load Instruction in Action – Example #2

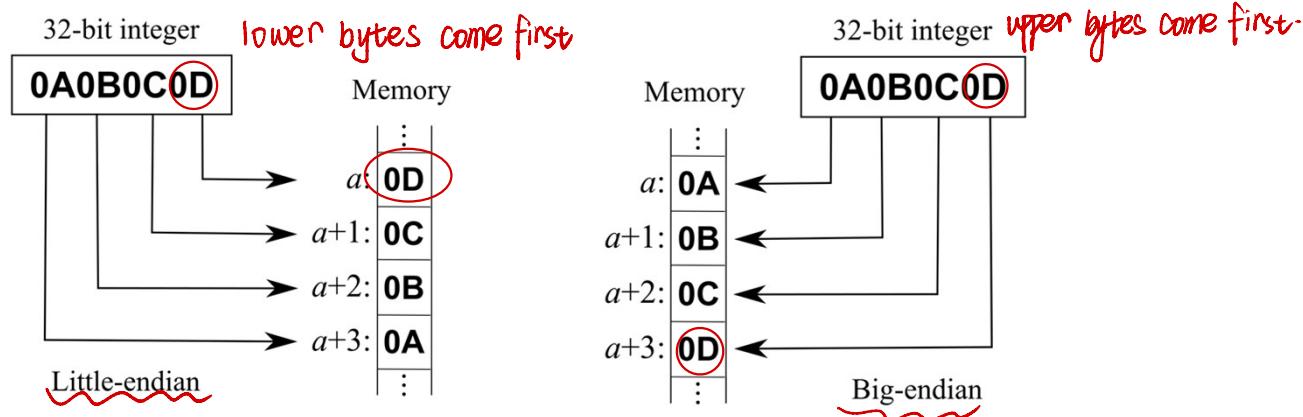
- LDURSW X3, [X4, #100] // load signed word
Retrieves 4-byte value from memory location (X4+100) and puts the result into X3 (sign extended)



Example mixes decimal and hex to keep the numbers easy to read
Recall that E = 1110 and thus is treated as a negative 2's complement number

Big Endian vs. Little Endian

- Endian-ness: ordering of bytes within a word
 - Little - increasing numeric significance with increasing memory addresses
 - Big – The opposite, most significant byte first
 - The Internet is big endian, x86 is little endian, LEG and ARMv8 can switch
 - But in general assume little endian. (Figures from Wikipedia)



L4_3 ARM ISA: Memory Instructions Examples

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

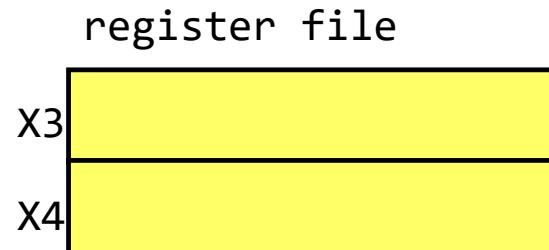
- Recognize the set of instructions for ARM Architecture (ISA) and be able to describe the operations and operands for instructions
 - LEGv8 subset
- Ability to create simple ARM assembly programs, e.g., using mathematical, logic, and memory operations

Example Code Sequence #1

ARM ISA

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

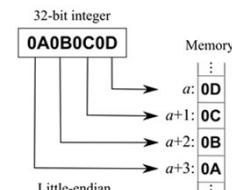
| ARM LEGv8 | |
|-----------|----------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



Memory
(each location is 1 byte)

| workspace | start | |
|-----------|-------|-----|
| | 0x02 | 100 |
| | 0x03 | 101 |
| | 0xFF | 102 |
| | 0x05 | 103 |
| | 0xC2 | 104 |
| | 0x06 | 105 |
| | 0xFF | 106 |
| | 0xE5 | 107 |

little endian

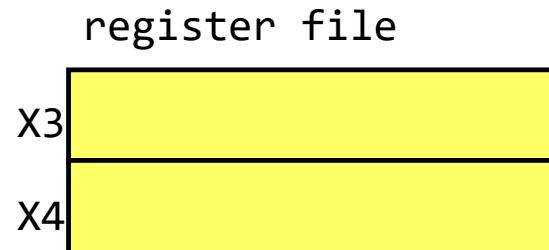


Example Code Sequence #1

ARM ISA

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

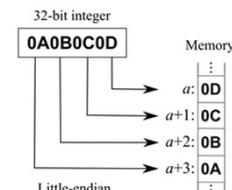
| ARM LEGv8 | |
|-----------|----------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



Memory
(each location is 1 byte)

| workspace | start | |
|-----------|-------|-----|
| | 0x02 | 100 |
| | 0x03 | 101 |
| | 0xFF | 102 |
| | 0x05 | 103 |
| | 0xC2 | 104 |
| | 0x06 | 105 |
| | 0xFF | 106 |
| | 0xE5 | 107 |

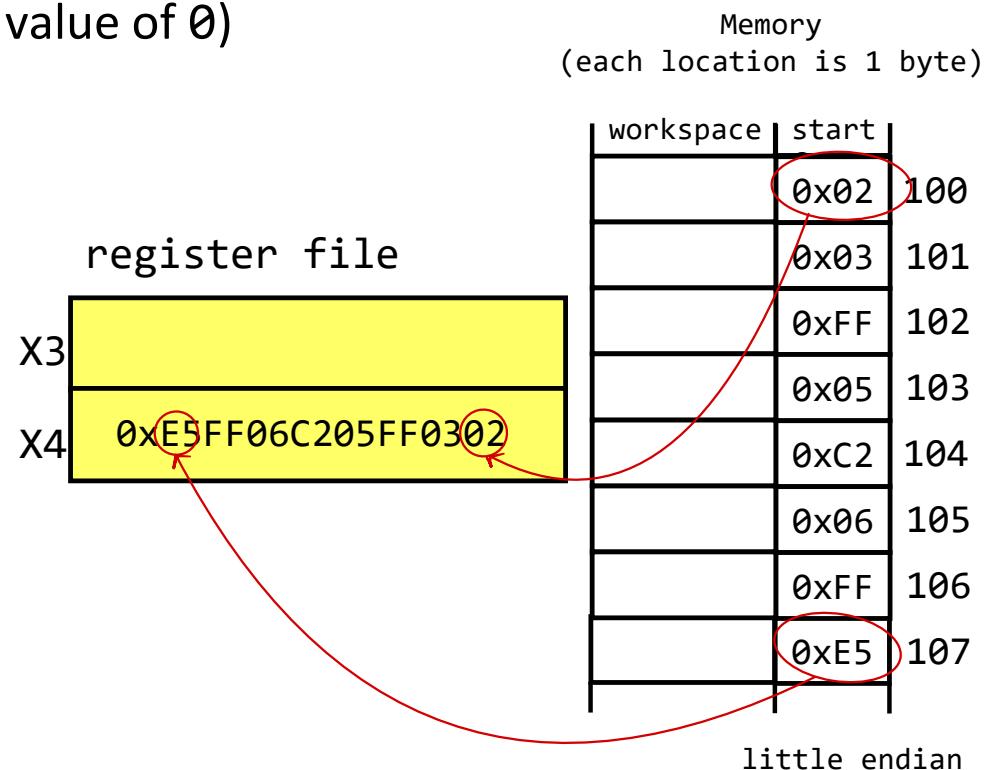
little endian



Example Code Sequence #1

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

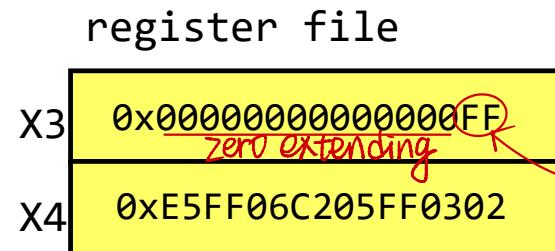
| ARM LEGv8 | |
|-------------|----------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



Example Code Sequence #1

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

| ARM LEGv8 | |
|--------------|-----------------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



Memory
(each location is 1 byte)

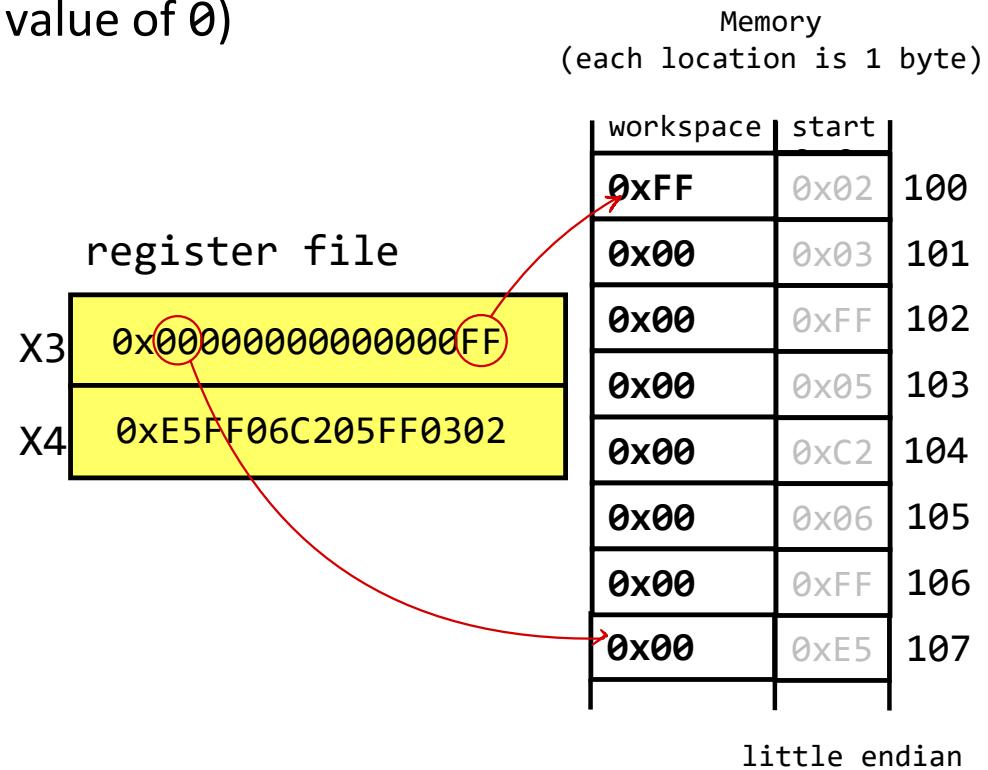
| workspace | start | |
|-----------|-------|-----|
| | 0x02 | 100 |
| | 0x03 | 101 |
| | 0xFF | 102 |
| | 0x05 | 103 |
| | 0xC2 | 104 |
| | 0x06 | 105 |
| | 0xFF | 106 |
| | 0xE5 | 107 |

little endian

Example Code Sequence #1

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

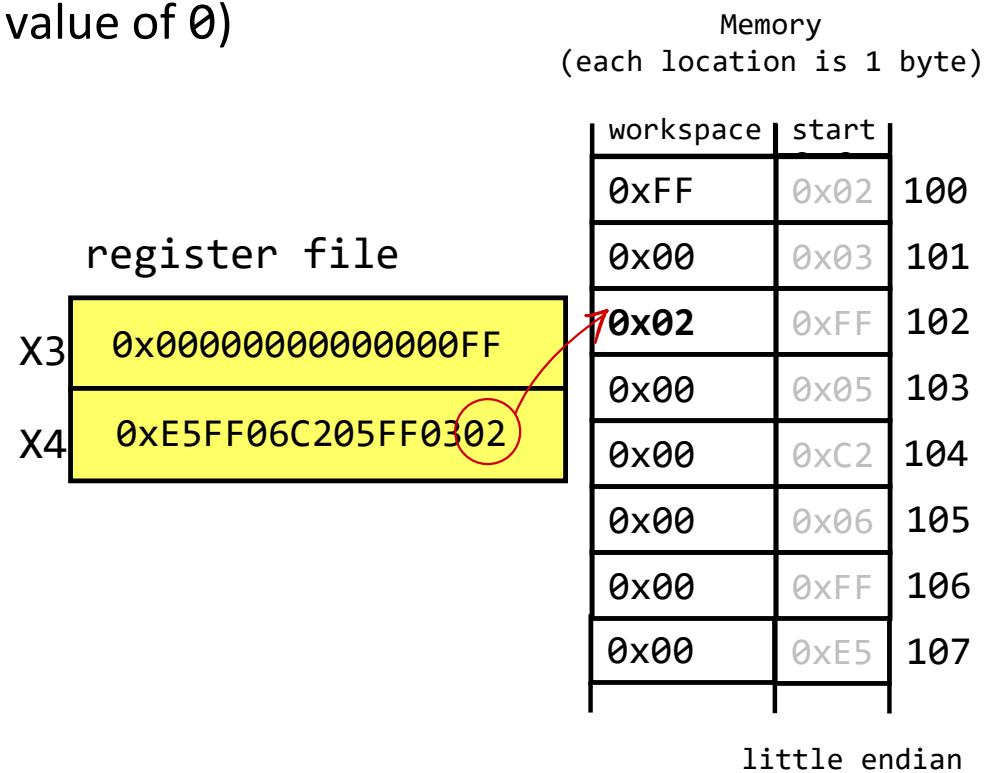
| ARM LEGv8 | |
|-------------|-----------------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



Example Code Sequence #1

- What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

| ARM LEGv8 | |
|--------------|-----------------------|
| LDUR | X4, [X5, #100] |
| LDURB | X3, [X5, #102] |
| STUR | X3, [X5, #100] |
| STURB | X4, [X5, #102] |



L4 4 C to Assembly

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

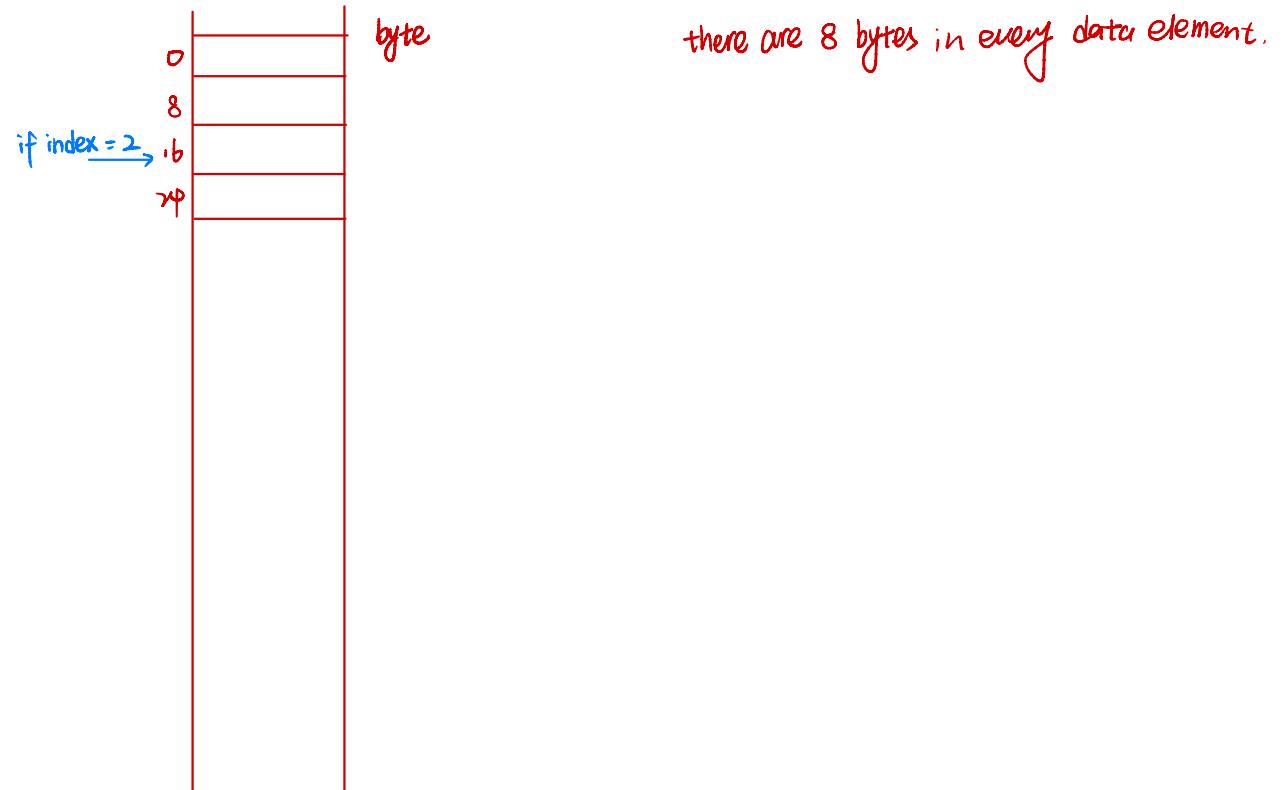
- Translate C-code *statements* to ARM assembly code
 - Break down complex C-code instructions into a series of assembly operations
 - Map variables to registers

Converting C to assembly – Example #1

Write ARM assembly code for the following C expression (the array holds 64-bit integers):

| Register to variable mapping |
|------------------------------|
| X1→a |
| X2→b |
| X3→i |
| X4→start address of names |

| C-code instruction |
|---------------------|
| a = b + names[i]; |



Converting C to assembly – Example #1

Write ARM assembly code for the following C expression (the array holds 64-bit integers):

| Register to variable mapping |
|------------------------------|
| X1→a |
| X2→b |
| X3→i |
| X4→start address of names |

| C-code instruction |
|---------------------|
| a = b + names[i]; |

| ARM LEGv8 |
|--|
| <i>Left shift logical</i> |
| LSL X5, X3, #3 // calculate array offset $i \cdot 8^3$ |
| ADD X6, X4, + X5 // calculate address of names[i] |
| LDUR X7, [X6, #0] // load names[i] |
| ADD X1, X2, + X7 // calculate b + names[i] |

Logistics

- There are two worksheets for lecture 4
 1. LEGv8 Assembly
 2. C to Assembly